9 Управление процессами

9.1 Процесс как системный объект

Процесс (вычислительный процесс, ***process***) – ассоциируется с выполняющейся программой.

Системный объект.

Идентификация – целочисленный идентификатор ***Process ID*** (***PID***), уникальный в пределах системы.
Генерация идентификаторов – счетчик; по достижении верхнего предела «заворачивается» к началу и выбирает освободившиеся значения.

В Unix-системах с поддержкой концепции ***вычислительных потоков*** (***thread***), потоки являются «облегченной версией» процессов с разделяемым адресным пространством, а место «настоящих» процессов занимают ***группы*** потоков.

9.2 Порождение и завершение процессов

Новый процесс – клон текущего (копии или совместное использование сегментов образа, новый идентификатор):

`fork()`

Взаимоотношения родительского (***parent***) и порожденного (***child***) процессов.

Семейство функций (системных вызовов) **`exec`** – замещение образа текущего процесса образом из файла (замена процесса с сохранением идентификатора и большинства атрибутов):

`execl()`
`execlp()`
`execle()`
`execv()`
`execvp()`
`execvpe()`

Завершение (штатное) процесса с заданным кодом возврата:

`exit()`

Принудительное завершение – как и другое управление процессом извне, реализовано через ***сигналы***.

9.3 Сигналы

Базовый (и одновременно простейший) IPC процессов для управления и взаимодействия.

Характеристика сигнала – только ***номер***. Назначение (смысл) сигнала однозначно определяется номером.

При отправке сигнала указывается его ***получатель*** (***адресат***).

Для получателя определены ***действия***, соответствующие каждому номеру сигналу.

Основные группы сигналов по назначению:

– извещения об аппаратных событиях, обычно ошибках (исключениях)

– извещения о событиях ввода-вывода

– извещения о событиях и изменении состояний в системе

– извещения о событиях других процессов

– управление процессом, в т.ч. отладка

– взаимодействие процессов

Возможные действия (обработка сигнала):

– нормальное завершение (normal termination) процесса-получателя

– аварийное завершение с сохранением дампа памяти (файл **`core`**)

– останов процесса, продолжение его выполнения (например, в ходе отладки)

– игнорирование

– пользовательский обработчик (подпрограмма процесса)

Количество поддерживаемых сигналов менялось по мере развития Unix-систем, наиболее часто используемый набор – около 30 (битовая маска рассчитана на 32 сигнала), но набор сигналов и номера некоторых из них могут быть различны в зависимости от семейства и версии Unix.

Набор сигналов, поддерживаемых конкретной системой:

```
kill -l
```

Например, набор сигналов в SuSE Linux:

Номер	Сигнал	Описание	Действие по умолчанию
1	**SIGHUP**	Обрыв терминальной линии («hang-up») для демонов используется как команда повторной инициализации и перезапуска	Завершение
2	**SIGINT**	«Прерывание» от консоли	Завершение

Номер	Сигнал	Описание	Действие по умолчанию
3	**SIGQUIT**	«Завершение» от консоли	Завершение с дампом
4	**SIGILL**	Некорректная инструкция ЦП	Завершение с дампом
5	**SIGTRAP**	Точка останова, трассировка	Завершение с дампом
6	**SIGABRT**	Завершение (вызов **abort()**)	Завершение с дампом
7	**SIGBUS**	«Ошибка шины» (ошибка доступа к памяти)	Завершение с дампом
8	**SIGFPE**	Ошибка вычислений с плавающей точкой	Завершение с дампом

Номер	Сигнал	Описание	Действие по умолчанию
9	SIGKILL	Безусловное завершение процесса	Завершение (не может быть переопределено)
10	SIGUSR1	Пользовательский сигнал	Завершение
11	SIGSEGV	Ошибка обращения к памяти, некорректный адрес, нарушение прав доступа	Завершение с дампом
12	SIGUSR2	Пользовательский сигнал	Завершение
13	SIGPIPE	Попытка записи в канал, который не может быть прочитан («broken pipe»)	Завершение

Номер	Сигнал	Описание	Действие по умолчанию
14	**SIGALRM**	Сигнал таймера (вызов **alarm()**)	Завершение
15	**SIGTERM**	Команда «мягкого» завершения процесса	Завершение
16	**SIGSTKFLT**	Ошибка стека сопроцессора (не исп.?)	Завершение
17	**SIGCHLD**	Завершение процесса-потомка	Игнорирование
18	**SIGCONT**	Продолжение остановленного процесса	Возобновление выполнения

Номер	Сигнал	Описание	Действие по умолчанию
19	SIGSTOP	Останов процесса для отладки и/или управления)	Останов (не может быть переопределено)
20	SIGTSTP	Останов от терминала (консоли)	Останов процесса
21	SIGTTIN	Консольный ввод для фонового процесса	Останов
22	SIGTTOU	Консольный вывод для фонового процесса	Останов
23	SIGURG	«Особое» (urgent) состояние сокета	Игнорирование

Номер	Сигнал	Описание	Действие по умолчанию
24	SIGXCPU	Превышение лимита использования ЦП	Завершение с дампом
25	SIGXFSZ	Превышение лимита размера файла (использования файловой системы)	Завершение с дампом
26	SIGVTALRM	Сигнал виртуального таймера	Завершение
27	SIGPROF	Сигнал «профилирующего» (profile) таймера	Завершение
28	SIGWINCH	Изменение размеров окна	Игнорирование

Номер	Сигнал	Описание	Действие по умолчанию
29	SIGIO	Событие (разрешение) ввода-вывода	Завершение
30	SIGPWR	Ошибка/сбой питания	Завершение
31	SIGSYS	Некорректный системный вызов	Завершение с дампом

Сигналы с номерами выше 32 считаются пользовательскими, предназначенными для взаимодействия процессов в реальном времени. Их диапазон задается системными константами **`SIGRTMIN`** и **`SIGRTMAX`**, обработка по умолчанию – игнорирование, индивидуальных имен не предусмотрено, порядок доставки несколько отличается от классических сигналов: возможность посылки нескольких сигналов без их «поглощения», наличие дополнительного параметра (целое число или указатель), соблюдение порядка доставки сигналов. Таким образом, эти сигналы по свойствам приближаются к ***сообщениям***.

34	SIGRTMIN
35	SIGRTMIN+1
36	SIGRTMIN+2
37	SIGRTMIN+3
38	SIGRTMIN+4
39	SIGRTMIN+5
40	SIGRTMIN+6
41	SIGRTMIN+7

42	SIGRTMIN+8
43	SIGRTMIN+9
44	SIGRTMIN+10
45	SIGRTMIN+11
46	SIGRTMIN+12
47	SIGRTMIN+13
48	SIGRTMIN+14
49	SIGRTMIN+15

50	SIGRTMAX-14
51	SIGRTMAX-13
52	SIGRTMAX-12
53	SIGRTMAX-11
54	SIGRTMAX-10
55	SIGRTMAX-9
56	SIGRTMAX-8
57	SIGRTMAX-7

58	SIGRTMAX-6
59	SIGRTMAX-5
60	SIGRTMAX-4
61	SIGRTMAX-3
62	SIGRTMAX-2
63	SIGRTMAX-1
64	SIGRTMAX

9.3.2 Порождение (отсылка) сигналов

Уровень командной строки (shell):

kill -*№_сигнала PID1 PID2 PID3* …

Уровень системных вызовов:

int kill(pid_t *pid*, int *sig_num*); – *сигнал процессу или группе процессов*

int killpg(int *pgid*, int *sig_num*); – *сигнал группе процессов*

С появлением многопоточности добавлены вызовы для посылки сигналов потокам:

int tkill(int *tid*, int *sig_num*); – *сигнал потоку текущей группы (устарел)*

int tgkill(int *tgid*, int *tid*, int *sig_num*); – *сигнал потоку заданной группы*

`int pthread_kill( pthread_t thread, int ssig_num);`
– *сигнал потоку*

Вызов **`kill()`** в этом случае посылает сигнал всему процессу, т.е. группе потоков (но не группе процессов).

Сигнал текущему процессу или текущему потоку многопоточной программы:

`int raise(int sig_num);`

9.3.3 Доставка сигналов

Доставка сигналов – асинхронная: процесс-получатель не обрабатывает сигналы немедленно после их доставки, но и не выбирает произвольно момент приема и обработки, поэтому его состояние (стадия выполнения) заранее не известно.

Ядро доставляет сигнал и инициирует его обработку только в моменты:
– возврат из режима ядра в режим задачи после выполнения системного вызова или обработки исключения;
– перед переходом в состояние ожидания («сна»), если это позволяет приоритет
– после выхода из состояния ожидания («сна»), если это позволяет приоритет

Таким образом, программа, которая не обращается к системным вызовам и не провоцирует исключения, не будет видеть отправленные ей сигналы (не считая «внешних» по отношению к ней исключений).

Явная проверка наличия сигналов (вызывается системой в перечисленные моменты «автоматической» обработки):

`issig()`

Системные вызовы явного ожидания сигналов:

`pause()` – *приостановка до получения любого сигнала*
`sigwait()` *и* **`sigsuspend()`** – *приостановка до получения любого сигнала по маске*
`wait()` *и* **`waitpid()`** – *приостановка до получения* **`SIGCHLD`** *(завершение потомка)*

9.3.4 Обработка сигналов

Предусмотрено три пути обработки поступивших сигналов:

– обработка по умолчанию (обработчик по умолчанию) – в соответствии с номером сигналов (см. выше);

– игнорирование – отсутствие реакции на сигнал независимо от его номера, но сигнал считается обработанным;

– пользовательский обработчик (функция в программе).

Текущее назначение сигналов – обработка по умолчанию, игнорирование или пользовательский обработчик – ***диспозиция***.

Сигналы **`SIGKILL`** и **`SIGSTOP`** могут быть обработаны только «по умолчанию», их диспозицию изменить нельзя[1].

[1] Можно, но это требует достаточно сложного дополнительного программирования на уровне ядра и соответствующих прав.

Обработчик сигнала (сигналов) – функция в программе, вызываемая автоматически при поступлении этих сигналов. Таким образом, обращение к функции происходит по принципу «***обратного вызова***» (***callback***). Технически возможно вызывать ее и явным образом из произвольного места программы (так делать нежелательно, более корректное решение – генерация сигнала «внутри» процесса вызовом **`raise()`**).

Один обработчик может обслуживать несколько сигналов, но каждому сигналу в пределах одного процесса назначается только один обработчик (новое назначение отменяет предыдущее).

Различают т.н. «ненадежные» (классические) и «надежные (стандарт POSIX) сигналы. Различие только в организации обработки, генерация и транспортировка сигналов едины.

9.3.5 Ненадежные (классические) сигналы

Более ранняя форма обработки, существовавшая в Unix изначально.

Формат функции-обработчика:

```
void mysighandler(int signum)
{
  switch (signum) {
  ... /*обработка конкретных сигналов*/
  }
}
```

Обычно бывает определен соответствующий процедурный тип:

```
typedef void (*sighandler_t)(int);
```

Функция (вызов) установки обработчика:

```
sighandler_t signal(
  int signum, sighandler_t new_handler);
```

Функция возвращает указатель на предыдущий обработчик этого сигнала или значение **SIG_ERR** в случае ошибки.

В качестве указателя на новый обработчик можно использовать константы: **SIG_DFL** (стандартный обработчик «по умолчанию») и **SIG_IGN** (игнорирование сигнала, «пустой» обработчик).

Важная особенность обработки сигналов – при передаче управления в обработчик диспозиция полученного сигнала сбрасывается на обработку «по умолчанию». Обработчик должен сам восстановить диспозицию вызовом **signal()**, если задача требует обрабатывать также и последующие сигналы. Сброс диспозиции – примитивный способ избежать неконтролируемого повторного вхождения в обработчик.

Два основных фактора ненадежности такой обработки:

1) Не предусмотрено избирательного блокирования сигналов на время активности обработчика. Повторное получение одного и того же сигнала после сброса его диспозиции, но до ее восстановления в обработчике приводит к обработке «по умолчанию» – например, нежелательному аварийное завершение процесса.

После восстановления диспозиции, но до завершения обработчика есть риск повторного получения того же сигнала и повторного входа в обработчик, который не обязательно написан реентерабельным. Та же проблема возникнет и при получении любого другого сигнала, которому назначен этот же обработчик.

2) Для «классических» (не «пользовательских») сигналов не предусмотрена очередь – при повторном получении сигнала с тем же номером до начала его обработки обработчик получит только один сигнал.

В любом случае, в обработчиках крайне нежелательно выполнять действия, требующие длительного выполнения или способные приводить к переходу в состояние ожидания или генерации новых сигналов.

9.3.6 Надежные (POSIX) сигналы

Исправляют только первую проблему ненадежности – повторное вхождение в обработчик.
Буферизация для «классических» сигналов по-прежнему отсутствует (в отличие от «пользовательских», но те не дублируют необходимые функции «классических»).

Два основных компонента решения:

1) Маска избирательной блокировки сигналов – содержит перечисление сигналов, обработка которых в данный момент запрещена и откладывается до момента разрешения.
Маска описывается типом **`segset_t`** («набор сигналов»).

Маска может применяться явно (соответствующим вызовом в произвольный момент) или автоматически (ассоциированная с обработчиком сигнала).

Явное применение маски:

```
int sigprocmask( int how,
  const sigset_t* set, sigset_t* oldset);
```

Параметр ***how*** определяет выполняемую операцию: полная замена маски (**SIG_SETMASK**), добавление к текущей маске блокировки сигналов из переданного набора set (**SIG_BLOCK**), удаление сигналов из текущей маски (**SIG_UNBLOCK**).

Создание, модификация, проверка маски: **sigemptyset()**, **sigfillset()**, **sigaddset()**, **sigismember()**.

Получение заблокированных (ожидающих обработки) сигналов:

```
int sigpending( sigset_t*);
```

Ожидание заданных сигналов:

```
int sigsuspend( const sigset_t*);
```

2) Усложнение описания обработчика – формат его вызова с дополнительными параметрами и структура **sigaction**, описывающая как функцию-обработчик, так и дополнительные характеристики обработки, включая автоматически применяемую маску блокировки сигналов:

```
struct sigaction_t {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int,siginfo_t*,void*);
  sigset_t sa_mask;
  int sa_flags;
  void (sa_restorer*)(void);
};
```

Здесь обработчики **`sa_handler()`** («старый» формат) и **`sa_sigaction()`** («новый» формат) – альтернативные, их не следует определить оба одновременно.
Флаги определяют дополнительные детали обработки. Флаг **`SA_SIGINFO`** – передача функции-обработчику трех аргументов вместо одного, т.е. будет использоваться «новый» формат вызова обработчика, следовательно должен быть определен обработчик **`sa_sigaction`**.

Установка «надежного» обработчика:

```
int sigaction( int signum,
  const struct sigaction* act,
  struct sigaction* oldact
);
```

Структура **`siginfo_t`** – дополнительная информация о сигнале: время, PID, UID и др.

9.4 Процессы-демоны

Типичный случай, когда обработка сигналов необходима – процессы-***демоны***: неинтерактивные, не имеющие возможности ввода-вывода через стандартные потоки, связанные с терминалом. Тем не менее, процесс-демон может пользоваться всеми прочими ресурсами системы: файловой системой, объектами IPC, сигналами.

Шаги превращения процесса в демон:

1) Создание нового процесса (вызов **`fork()`**).
Результат:
– процесс-потомок точно не является лидером группы
– процесс-родитель может быть нормально завершен, что имеет значение для его родителя
– после завершения процесса-родителя потомок остается в ведении процесса **`init`**.

2) Процесс становится лидером нового сеанса и группы (**`setsid()`**, **`setpgrp()`**)
Результат:
– «отвязывание» от управляющего терминала и текущего «родительского» сеанса

3) (опционально) Повторно создание нового процесса
Результат:
– новый процесс – «non-session group leader»

4) (опционально) Отключение реакции по умолчанию на связанные с терминалом сигналы: **`SIGTTIN`**, **`SIGTTOU`**, **`SIGTSTP`**; отключение прочих сигналов, способных прервать процесс
Результат:
– независимость от событий терминала (при правильном выполнении других шагов он все равно должен быть отключен)

6) Изменение текущего директория на корневой (он сохранится при любых обстоятельствах, а другие могут быть удалены, размонтированы, смонтированы иным образом)

7) (опционально) Сброс унаследованных масок (**`umask()`**)
Результат:
– полный контроль над записываемыми файлами

8) Закрытие как минимум дескрипторов стандартных потоков ввода-вывода, как максимум всех открытых дескрипторов
Результат:
– освобождение дескрипторов, предохранение от последствий ошибочного обращения к операциям ввода-вывода

9) (опционально) Установка стандартных дескрипторов на файлы, например /dev/null
Результат:
– возможность сохранения в коде демона функций ввода-вывода, рассчитанных на стандартные потоки, без возникновения ошибок при обращении к ним

10) Настройка необходимых обработчиков сигналов, в т.ч. **`SIGHUP`**, открытие необходимых файлов и устройств, подключение к необходимым объектам IPC
Результат:
– программный интерфейс для взаимодействия с демоном

Пример процесса-демона:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include "timer.h"
#include "logger.h"
#define PERIOD 10


static unsigned long n_count;
static int b_exitflag;


/* -- Signal handler (old-style signal handling) --
*/
```

```
void SigHandler( int n_signal)
{
  switch (n_signal) {
   case SIGHUP:
    n_count = 0; //re-initializing must be here!
    break;
   case SIGTERM:
    b_exitflag = 1; //flag to exit the process
    break;
   default:
    break;
  }
  signal( n_signal, SigHandler); //restore
disposition
  return;
}
```

```
/* -- Main module entry point -- */
int main( int argc, char* argv[])
{
 char sz_strbuf[512];
 pid_t n_pid;
 unsigned long n_lastcount = n_count;
//- initializing
 n_count = 0;
 n_pid = getpid();
 strcpy( sz_strbuf, argv[0]); strcat( sz_strbuf,
".log");
 LogOpen( sz_strbuf, n_pid);
 LogPost( "Starting", 0);
//- installing daemon
//- step 1
 n_pid = fork();
```

```
  switch (n_pid) {
   case -1: //the parent is here, an error occured
    LogPost( "fork() error!", 0);
    LogClose();
    exit( -1);
   case 0: //the child is here
    n_pid = getpid();
    LogPost( "The process was switched", n_pid);
//with new PID
    break;
   default: //parent is here, child was started
    LogClose();
    exit( 0);
  }
//- step 2 (child is here)
  setsid(); //new personal session
```

```
//- step 3
  n_pid = fork();
  switch (n_pid) {
   case -1: //parent (first child) is here, error
    LogPost( "fork() error!", 0);
    LogClose();
    exit( -1);
   case 0: //the child (second child) is here
    n_pid = getpid();
    LogPost( "The process was double switched",
n_pid); //with new PID
    break;
   default: //parent (first child) is here,
//and second child was started
    LogClose();
    exit( 0);
  }
```

```
//- step 4 (the second child is here)
  signal( SIGTTOU, SIG_IGN);
  signal( SIGTTIN, SIG_IGN);
  signal( SIGTSTP, SIG_IGN);
  signal( SIGHUP, SigHandler);
  signal( SIGTERM, SigHandler);
//- step 5
  chdir( "/");
  fclose( stdin); fclose( stdout); fclose( stderr);
//- main loop
  LogPost( "Waiting for events", 0);
  while (! b_exitflag) {
    sleep( PERIOD);
    if (n_count < n_lastcount) { //detect re-
initializing (indirectly)
      LogPost( "Re-initialization occured!", 0);
    }
```

```
    n_count += PERIOD;
    n_lastcount = n_count;
    sprintf( sz_strbuf, "Counter: %ld seconds",
n_count);
    LogPost( sz_strbuf, 0);
   }
//- closing, then exit
  LogPost( "Exiting", 0);
  LogClose();
  exit( 0);
}
```